

Better Space Bounds for Parameterized Range Majority and Minority

Djamal Belazzougui¹, Travis Gagie², and Gonzalo Navarro³

¹ LIAFA, University of Paris Diderot (Paris 7)

² Department of Computer Science and Engineering, Aalto University

³ Department of Computer Science, University of Chile

Abstract

We consider how to store an array of length n such that we can quickly list all the distinct elements whose relative frequencies in a given range are greater than a given threshold τ . We give the first linear-space data structure that answers queries in worst-case optimal $\mathcal{O}(1/\tau)$ time, assuming τ is pre-specified. We then consider the more challenging case when τ is given at query time. The best previous bounds for this problem are $\mathcal{O}(n \lg \sigma)$ space and $\mathcal{O}(1/\tau)$ time, where $\sigma \leq n$ is the number of distinct elements in the array. We show how to use either $n \lg \lg \sigma + \mathcal{O}(n)$ space and $\mathcal{O}(1/\tau)$ time, linear space and $\mathcal{O}((1/\tau) \lg \lg(1/\tau))$ time, or compressed space and $\mathcal{O}((1/\tau) \lg \lg \sigma)$ time. We also consider how to find a single element whose relative frequency in the range is below the threshold (range minority), or determine that no such element exists. We show how to modify a linear-space data structure for this second problem by Chan et al. (2012) so that it uses compressed space and almost the same time $\mathcal{O}(1/\tau)$. We obtain in passing some results of independent interest, for example density-sensitive query time for one-dimensional range counting.

1998 ACM Subject Classification E.1 Data Structures

Keywords and phrases range queries; (in)frequent elements; compressed data structures

1 Introduction

Parameterized range-majority (PRMaj) queries have been the subject of recent research, e.g., [19, 24, 12] and the papers mentioned below. For this problem we are asked to preprocess an array A such that later, given i and j , we can quickly list all the distinct elements whose relative frequency in $A[i..j]$ is above a given threshold τ . Karpinski and Nekrich [18] showed how, if τ is given to us at the same time as A , then we can store A in $\mathcal{O}(n/\tau)$ space on a RAM with $\Omega(\lg n)$ -bit words and answer queries in $\mathcal{O}((1/\tau)(\lg \lg n)^2)$ time. Durocher et al. [11] reduced the space bound to $\mathcal{O}(n \lg(1/\tau + 1))$ words and the time bound to $\mathcal{O}(1/\tau)$. Since there could be $1/\tau$ elements to list, this time bound is worst-case optimal. Gagie et al. [15] showed how to store A either in $\mathcal{O}(n(H + 1))$ words with query time $\mathcal{O}(1/\tau)$ or in $\mathcal{O}(n(H + 1))$ bits with query time $\mathcal{O}((1/\tau) \lg \lg \sigma)$, where $H \leq \lg \sigma$ is the entropy of the distribution of elements in A and $\sigma \leq n$ is the number of distinct elements in A . Throughout, we assume the distinct elements in A are $\{1, \dots, \sigma\}$; otherwise, they can be mapped to that set using a simple translation array. Gagie et al. also showed that, for their first data structure, τ can be given with i and j at query time. That is, we can store A in $\mathcal{O}(n(H + 1))$ words such that later, given i , j and τ , in $\mathcal{O}(1/\tau)$ time we can list all the distinct elements that occur at least $\tau(j - i + 1)$ times in $A[i..j]$. Chan et al. [10] independently gave another data structure that can handle variable τ , but it takes $\mathcal{O}(n \lg n)$ words and the same time. These are the only previous solutions for the important case of variable τ .



© D. Belazzougui, T. Gagie and G. Navarro;
licensed under Creative Commons License ND



Leibniz International Proceedings in Informatics
LIPIC Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

■ **Table 1** Results for the PRMaj problem on an array of length n containing σ distinct elements, in which the distribution of elements has entropy $H \leq \lg \sigma$. The solutions in lines 3 and 9 are actually compressed.

Source	Space (words)	Time (order)	variable τ
Karpinski and Nekrich [18]	$\mathcal{O}(n/\tau)$	$(1/\tau)(\lg \lg n)^2$	no
Durocher et al. [11]	$\mathcal{O}(n \lg(1/\tau))$	$1/\tau$	no
Gagie et al. [15]	$\mathcal{O}(n)$	$(1/\tau) \lg \lg \sigma$	no
Theorem 2	$\mathcal{O}(n)$	$1/\tau$	no
Gagie et al. [15]	$\mathcal{O}(n(H+1))$	$1/\tau$	yes
Chan et al. [10]	$\mathcal{O}(n \lg n)$	$1/\tau$	yes
Theorem 5	$n \lg \lg \sigma + \mathcal{O}(n)$	$1/\tau$	yes
Theorem 11	$\mathcal{O}(n)$	$(1/\tau) \lg \lg(1/\tau)$	yes
Theorem 8	$n + o(n)$	$(1/\tau) \lg \lg \sigma$	yes

In this paper we start by giving the first linear-space data structure for the original PRMaj problem (i.e., when τ is pre-specified at construction time) that answers queries in worst-case optimal $\mathcal{O}(1/\tau)$ time. We then turn our attention to the more challenging case when τ is given at query time. We show how to store $n \lg \lg \sigma + \mathcal{O}(n)$ words such that later, given i, j and τ , in $\mathcal{O}(1/\tau)$ time we can list all the distinct elements that occur at least $\tau(j-i+1)$ times in $A[i..j]$. Note the space is near-linear and the time is still worst-case optimal. We then give another solution that represents A using only $nH + o(n) \lg \sigma$ bits, and solves queries in the slightly higher time $\mathcal{O}((1/\tau) \lg \lg \sigma)$. This the first *succinct* (i.e., $n + o(n)$ words) solution to the problem (for fixed or variable τ), and is also *compressed*. In the full version of this paper we will reduce its space bound further, to $nH + o(n(H+1))$ bits, making our solution *fully compressed*. Finally, we give a third solution with $\mathcal{O}(n)$ words of space and time $\mathcal{O}((1/\tau) \lg \lg(1/\tau))$. Table 1 summarizes previous results and our own ones. In our time complexities, $1/\tau$ actually stands for $\min(1/\tau, \sigma)$. To measure our space in words we assume the computer word contains exactly $\lceil \lg n \rceil$ bits.

Chan et al. [10] also introduced the related problem of parameterized range minority (PRMin). For this problem we are asked to preprocess A such that, given i, j and τ , we can quickly find a single element that occurs in $A[i..j]$ but at most $\tau(j-i+1)$ times, or determine that no such element exists. They gave a data structure that takes $\mathcal{O}(n)$ words and $\mathcal{O}(1/\tau)$ time. In this paper we improve the space bound to $(1+\epsilon)nH + \mathcal{O}(n)$ bits for any constant $\epsilon > 0$. We also give another solution that takes only $nH + \mathcal{O}(n) + o(nH)$ bits but $\mathcal{O}((1/\tau)\alpha(n))$ time, where $\alpha(n)$ is the very slowly-growing inverse Ackermann function of n . This data structure is both compressed and the first succinct solution for PRMin. In the full version of this paper we will reduce the space bound further, to $nH + o(n(H+1))$ bits, making our solution fully compressed.

Along the way we obtain some results of independent interest. Perhaps the most relevant one is a density-sensitive data structure for the one-dimensional range counting problem: Given n points in a discrete universe, we can build a linear-space data structure that can compute the number occ of points inside any range $[i, j]$ in time $\mathcal{O}\left(\lg \lg \frac{j-i+1}{occ+1}\right)$.

2 Parameterized Range Majority

Our data structures are based on the well-studied operations of access, rank and select. If S is a sequence (e.g., a one-dimensional array) then $S.\text{rank}_a(k)$ is the number of occurrences of a in $S[1..k]$ and $S.\text{select}_a(k)$ is the position of the k th occurrence of a in S . The data structures we use for access, rank and select in this section are by Pătraşcu [22] for binary sequences, by Ferragina et al. [13] for sequences over polylogarithmic alphabets, and by Barbay et al. [3] for general sequences (there are slightly better choices for the last two, but these simple ones are sufficient for us). For the first two cases we can support the three operations in constant time and space $nH + o(n)$ bits, where H is the zero-order entropy of S (for bitmaps, the $o(n)$ term is as good as $\mathcal{O}(n/\lg^c n)$ for any constant c). For large alphabets, we use $nH + o(n(H+1))$ bits and can have access or select in constant time, and the other two operations in time $\mathcal{O}(\lg \lg \sigma)$. Moreover, we can add $\mathcal{O}(n \lg \lg \sigma)$ bits in order to support in constant time partial rank queries, which are of the form $S.\text{rank}_{S[k]}(k)$ [4].

The main idea in this section is to precompute, for a sampling of the possible intervals, the frequent elements in those intervals. Then we process the query range using a few sampled intervals that contain it. Interval sizes and frequent elements are chosen so that the latter are sufficiently few to allow us checking one by one whether they are a majority in the query range. If we spend sufficient space to spot the first occurrence of those elements in the query range, then we achieve optimal time by checking them using only select and partial rank, in constant time. If we use less space, instead, we can only spot one arbitrary occurrence in the query range, and need to resort to the slower rank operation.

2.1 Linear space and optimal time for pre-specified τ

Let $C[1..n]$ be the array in which $C[k] = \text{select}_{A[k]}(\text{rank}_{A[k]}(k) - 1)$ if $A[k]$ is not the first occurrence of that distinct element in A , or 0 if it is. Notice that, if $i \leq k \leq j$, then $C[k] < i$ if and only if $A[k]$ is the first occurrence of that distinct element in $A[i..j]$. Muthukrishnan [21] showed that, if we store a data structure supporting range-minimum queries (RMQs) on C in $\mathcal{O}(1)$ time then later, given i and j , we can list the positions of the first occurrences in $A[i..j]$ of the distinct elements in $A[i..j]$, in time proportional to the number of such elements. Sadakane [23] adapted Muthukrishnan's approach to the case when we have access to A but not C and the RMQ data structure returns only the position of the range minimum in C , in $\mathcal{O}(1)$ time. Fischer [14] showed that such an RMQ data structure takes only $2n + o(n)$ bits.

► **Lemma 1.** *Suppose we are given A , τ and a value $b \leq \lfloor \lg n \rfloor$. Then we can store $\mathcal{O}(n)$ more bits such that later, given i and j with $\lfloor \lg(j-i+1) \rfloor = b$, in $\mathcal{O}(1/\tau)$ time we can build a list of $\mathcal{O}(1/\tau)$ positions that includes the position of the first occurrence in $A[i..j]$ of each distinct element that occurs at least $\tau(j-i+1)$ times in $A[i..j]$.*

Proof. We store a bitvector $B[1..n]$ with $B[k] = 1$ if $A[k]$ occurs at least $\tau 2^b$ times in $A[k..k + 2^{b+1} - 1]$, and $B[k] = 0$ otherwise. Let A' be the array consisting of elements of A indicated by 1s in B — so $A'[k] = A[B.\text{select}(k)]$ — and let C' be the array in which $C'[k] = \text{select}_{A'[k]}(\text{rank}_{A'[k]}(k) - 1)$ if $A'[k]$ is not the first occurrence of that distinct element in A' , or 0 if it is. We store neither A' nor C' explicitly (although we can use A and B to support access to A' in $\mathcal{O}(1)$ time) but only an RMQ data structure over C' . Together, B and this RMQ data structure take $\mathcal{O}(n)$ bits.

Given i and j , we compute $i' = B.\text{rank}(i-1) + 1$ and $j' = B.\text{rank}(j)$ and use Sadakane's algorithm and the RMQ data structure for C' to list the positions of the first occurrences in

$A'[i'..j']$ of the distinct elements in $A'[i'..j']$. Once we have these positions, we use select on B to map them back to positions in A . Notice that any distinct element in $A'[i'..j']$ occurs at least $\tau 2^b$ times in $A[i..j + 2^{b+1} - 1]$, so there are $\mathcal{O}(1/\tau)$ of them. It follows that we use a total of $\mathcal{O}(1/\tau)$ time to build a list of $\mathcal{O}(1/\tau)$ positions in A . The first occurrence in $A[i..j]$ of any element that occurs at least $\tau(j - i + 1)$ times in $A[i..j]$, is marked by a 1 in B ; therefore, that position is in our list. \blacktriangleleft

► **Theorem 2.** *Given A and τ , we can store A in linear space such that later, given i and j , in $\mathcal{O}(1/\tau)$ time we can list all the distinct elements that occur at least $\tau(j - i + 1)$ times in $A[i..j]$.*

Proof. We apply Lemma 1 to A for each value of b between 0 and $\lfloor \lg n \rfloor$, which takes a total of $\mathcal{O}(n \lg n)$ bits, so $\mathcal{O}(n)$ words. We also store a single linear-space data structure supporting access, select and partial rank on A in $\mathcal{O}(1)$ time. Given i and j , in $\mathcal{O}(1/\tau)$ time we build a list of $\mathcal{O}(1/\tau)$ positions in $A[i..j]$ that includes the position of the first occurrence in $A[i..j]$ of each distinct element that occurs at least $\tau(j - i + 1)$ times in $A[i..j]$. For each position k in this list, we return $A[k]$ if $A.\text{select}_{A[k]}(A.\text{rank}_{A[k]}(k) + \lceil \tau(j - i + 1) \rceil - 1) \leq j$. Repetitions are avoided by marking a bitmap of length $\sigma \leq n$. \blacktriangleleft

In the full version of this paper we will give slightly tighter bounds. For example, we do not need to consider $b < \lg(1/\tau)$ because, if $j - i + 1 = \mathcal{O}(1/\tau)$, then we can afford to run on $A[i..j]$ a linear-time algorithm by Misra and Gries [20] that lists the distinct elements that occur at least $\tau(j - i + 1)$ times. Taking advantage of this, we can reduce our space bound to $\mathcal{O}(n(\lg n - \lg(1/\tau))) = \mathcal{O}(n \lg(\tau n))$ bits, or $\mathcal{O}(n \lg(\tau n)/\lg n)$ words, plus compressed space to support access, select and partial rank on A .

2.2 Near-linear space and optimal time

We now turn our attention to the more challenging case of when τ is given at query time. We reuse some of the techniques from Section 2.1 to reduce the space bound for this version of the problem as well.

► **Lemma 3.** *Suppose we are given A and C and a value $b \leq \lfloor \lg n \rfloor$. Then we can store $n \lg \lg \sigma + \mathcal{O}(n)$ more bits such that later, given i , j and τ with $\lfloor \lg(j - i + 1) \rfloor = b$, in $\mathcal{O}(1/\tau)$ time we can build a list of $\mathcal{O}(1/\tau)$ positions that includes the position of the first occurrence in $A[i..j]$ of each distinct element that occurs at least $\tau(j - i + 1)$ times in $A[i..j]$.*

Proof. We store an RMQ data structure over C , which takes $\mathcal{O}(n)$ bits. If $1/\tau = \Omega(\sigma)$, then we can simply list the positions of the first occurrences in $A[i..j]$ of all the distinct elements in $A[i..j]$, using Muthukrishnan's algorithm in $\mathcal{O}(\sigma)$ time. Therefore, we need consider only the case when $1/\tau \leq \sigma$.

Let $T[1..n]$ be the array in which $T[k]$ is the smallest integer $t \leq \lceil \lg \sigma \rceil$ such that $A[k]$ occurs at least $2^b/2^t$ times in $A[k..k + 2^{b+1} - 1]$, or ∞ if there is no such integer. We can store T in $n \lg \lg \sigma + o(n)$ bits with an instance of Ferragina et al.'s [13] multiary wavelet tree and support access, rank and select on T in $\mathcal{O}(1)$ time since T is over a polylogarithmic alphabet. For $t \leq \lceil \lg \sigma \rceil$, let A_t and C_t be the arrays whose lengths are the number of occurrences of t in T and in which $A_t[p] = A[T.\text{select}_t(p)]$ and $C_t[p] = C[T.\text{select}_t(p)]$. Notice that we can support access to A_t and C_t in $\mathcal{O}(1)$ time using select on T and access to A and C , so we need not store A_t and C_t explicitly. For each $t \leq \lceil \lg \sigma \rceil$, we store an RMQ data structure over C_t . Since the total length of $C_0, \dots, C_{\lceil \lg \sigma \rceil}$ is at most n , all the RMQ data structures together take a total of $\mathcal{O}(n)$ bits.

Given i, j and $\tau \geq 1/\sigma$, for each value $t \leq \lceil \lg(1/\tau) \rceil$ we compute $i' = T.\text{rank}_t(i-1) + 1$ and $j' = T.\text{rank}_t(j)$, then list the positions of the first occurrences in $A_t[i'..j']$ of the distinct elements in $A_t[i'..j']$. Once we know the position p of the first occurrence of a distinct element in $A_t[i'..j']$, we can use $k = T.\text{select}_t(p)$ to find the corresponding position in $A[i..j]$. Each distinct element in $A_t[i'..j']$ occurs at least $2^b/2^t$ times in $A[k..k + 2^{b+1} - 1]$ for some $i \leq k \leq j$, which is contained in $A[i..j + 2^{b+1} - 1]$, and thus there can be at most $4 \cdot 2^t$ of them because $2^b \leq j - i + 1 < 2^{b+1}$. It follows that we list a total of $\sum_{t=0}^{\lceil \lg(1/\tau) \rceil} 4 \cdot 2^t = \mathcal{O}(1/\tau)$ positions for all values t together.

Any distinct element that occurs at least $\tau(j - i + 1)$ times in $A[i..j]$, occurs at least $2^b/2^{\lceil \lg(1/\tau) \rceil} \leq \tau 2^b \leq \tau(j - i + 1)$ times in $A[i..j]$, so if $k \geq i$ is its leftmost occurrence in the interval, it occurs at least $2^b/2^{\lceil \lg(1/\tau) \rceil}$ times in $A[k..j]$, which contains $A[k..k + 2^{b+1} - 1]$. Thus, we list the position of the first occurrence in $A[i..j]$ of each such distinct element. ◀

► **Corollary 4.** *Given A and C , we can store $n \lg \lg \sigma + \mathcal{O}(n)$ more words such that later, given i, j and τ , in $\mathcal{O}(1/\tau)$ time we can build a list of $\mathcal{O}(1/\tau)$ positions that includes the position of the first occurrence in $A[i..j]$ of each distinct element that occurs at least $\tau(j - i + 1)$ times in $A[i..j]$.*

Proof. We apply Lemma 3 for all values of b between 0 and $\lfloor \lg n \rfloor$, which takes a total of $n \lg n \lg \lg \sigma + \mathcal{O}(n \lg n)$ bits or $n \lg \lg \sigma + \mathcal{O}(n)$ words. ◀

► **Theorem 5.** *We can store A in $n \lg \lg \sigma + \mathcal{O}(n)$ words such that later, given i, j and τ , in $\mathcal{O}(1/\tau)$ time we can list all the distinct elements that occur at least $\tau(j - i + 1)$ times in $A[i..j]$.*

Proof. We apply Corollary 4 to A and store $\mathcal{O}(n)$ words such that we can support access, partial rank and select on A in $\mathcal{O}(1)$ time (using, say, two copies of the structure of Barbay et al. [3] plus the partial rank structure [4]). Given i, j and τ , we build a list of $\mathcal{O}(1/\tau)$ positions that includes the position of the first occurrence in $A[i..j]$ of each distinct element that occurs at least $\tau(j - i + 1)$ times in $A[i..j]$. For each position k in this list, we list $A[k]$ if $A.\text{select}_{A[k]}(A.\text{rank}_{A[k]}(k) + \lceil \tau(j - i + 1) \rceil - 1) \leq j$. Repetitions are avoided by marking a bitmap of length $\sigma \leq n$. ◀

2.3 Compressed space and near-optimal time

► **Lemma 6.** *Suppose we are given values $b \leq \lfloor \lg n \rfloor$ and $t \leq \lceil \lg n \rceil$ and a data structure supporting access and rank on A in $\mathcal{O}(\lg \lg \sigma)$ time. Then we can store $\mathcal{O}\left(\frac{n 2^t (b-t)}{2^b} + \frac{n}{\lg^3 n}\right)$ more bits such that later, given i, j and τ with $\lfloor \lg(j - i + 1) \rfloor = b$ and $\lceil \lg(1/\tau) \rceil = t$, in $\mathcal{O}((1/\tau) \lg \lg \sigma)$ time we can list all the distinct elements that occur at least $\tau(j - i + 1)$ times in $A[i..j]$.*

Proof. We divide A into blocks of length 2^{b-1} and store a bitvector $B[1..n]$ in which $B[k] = 1$ if, first, $A[k]$ is the first or last occurrence of that distinct element in that block and, second, $A[k]$ occurs at least $2^b/2^t$ times in $A[k - 2^{b+1}..k + 2^{b+1}]$. At most $9 \cdot 2^t$ elements in any block $A[\ell.. \ell + 2^{b-1} - 1]$ are marked by 1s in B , because at most $9 \cdot 2^{t-1}$ distinct elements occur at least $2^b/2^t$ times (each) in $A[\ell - 2^{b+1}.. \ell + 2^{b-1} - 1 + 2^{b+1}]$. It follows that we can store B in $\mathcal{O}\left(\frac{n 2^t (b-t)}{2^b} + \frac{n}{\lg^3 n}\right)$ bits using Pătraşcu's [22] structure (recall that $nH = \mathcal{O}(m \lg \frac{n}{m})$ on a bitmap with m 1s; here $m \leq 9 \cdot 2^t n/2^{b-1}$).

Given i, j and τ , we find all the distinct elements in $A[i..j]$ marked by 1s in B . Since $j - i + 1 < 2^{b+1}$, $A[i..j]$ overlaps at most 5 blocks, thus there are at most $45 \cdot 2^t = \mathcal{O}(1/\tau)$

marked elements. For each such element a , we compute $A.\text{rank}_a(j) - A.\text{rank}_a(i-1)$ and output a if that difference is at least $\tau(j-i+1)$. This takes a total of $\mathcal{O}((1/\tau) \lg \lg \sigma)$ time.

To see that our list is complete, suppose we should list an element a . Notice $A[i..j]$ must include either the beginning or the end of the block containing the first occurrence of a in $A[i..j]$. If it includes the beginning, then that occurrence of a is marked; otherwise, the last occurrence of a in the block is both in $A[i..j]$ and marked. Repeated elements are avoided as before, using σ extra bits. \blacktriangleleft

Notice that, even if some occurrence of a in $A[i..j]$ must be marked by a 1 in B , it may not be the first such occurrence. This is why we use rank here, and not the faster combination of partial rank and select that we used in the proof of Theorem 5.

► **Corollary 7.** *Suppose we are given a value $t \leq \lceil \lg n \rceil$ and a data structure supporting access on A in $\mathcal{O}(1)$ time and rank on A in $\mathcal{O}(\lg \lg \sigma)$ time. Then we can store $\mathcal{O}\left(\frac{n \lg \lg \lg \sigma}{\lg \lg \sigma} + \frac{n}{\lg^2 n}\right)$ more bits such that later, given i, j and τ with $\lceil \lg(1/\tau) \rceil = t$, in $\mathcal{O}((1/\tau) \lg \lg \sigma)$ time we can list all the distinct elements that occur at least $\tau(j-i+1)$ times in $A[i..j]$.*

Proof. If $j-i-1 = \mathcal{O}((1/\tau) \lg \lg \sigma)$, then we can afford to run Misra and Gries' [20] algorithm. Therefore, we need apply Lemma 6 only for values of b between $\lfloor \lg(2^t \lg \lg \sigma) \rfloor$ and $\lfloor \lg n \rfloor$, which takes a total of $\mathcal{O}\left(\frac{n \lg \lg \lg \sigma}{\lg \lg \sigma} + \frac{n}{\lg^2 n}\right)$ bits. \blacktriangleleft

► **Theorem 8.** *We can store A in $nH + o(n) \lg \sigma$ bits such that later, given i, j and τ , in $\mathcal{O}((1/\tau) \lg \lg \sigma)$ time we can list all the distinct elements that occur at least $\tau(j-i+1)$ times in $A[i..j]$. The structure also offers constant-time access to A and $\mathcal{O}(\lg \lg \sigma)$ -time rank and select.*

Proof. We store A in a data structure by Barbay et al. [3, 2] that takes $nH + o(n(H+1))$ bits and supports access in constant time, and rank and select in $\mathcal{O}(\lg \lg \sigma)$ time. If $1/\tau \geq \sigma$, then we can afford to compute $A.\text{rank}_a(j) - A.\text{rank}_a(i-1)$ for each distinct element a in A . Therefore, we need apply Corollary 7 only for values of t between 0 and $\lceil \lg \sigma \rceil$, which takes a total of $\mathcal{O}\left(\frac{n \lg \sigma \lg \lg \lg \sigma}{\lg \lg \sigma} + \frac{n}{\lg n}\right)$ bits. This is $o(n) \lg \sigma$ unless $\sigma = \mathcal{O}(1)$, in which case we can compute the frequency of each distinct symbol in the range, in a total of $\mathcal{O}(\sigma \lg \lg \sigma) = \mathcal{O}(1)$ time. \blacktriangleleft

In the full version of this paper we will reduce the space bound in Theorem 8 to $nH + o(n(H+1))$ bits. For those reviewers who are curious, we have included a sketch of the ideas in Appendix A.

2.4 Linear space and nearer-optimal time

We now give a third tradeoff by building on the same solution of Section 2.3. The idea is to replace the compressed representation of A by a linear-space one that allows us to count the number of occurrences occ of any element in any range $A[i..j]$ in time $\mathcal{O}\left(\lg \lg \left(\frac{j-i+1}{\text{occ}+1}\right)\right)$. To achieve it, we start describing a *one-dimensional range counting data structure*, which handles n points in $[1, U]$ and can count the number occ of points in any range $[i, j]$ within the same time. Such structures have independent interest.

► **Theorem 9.** *Given n points in the discrete universe $[1, U]$, there exists a data structure using $\mathcal{O}(n \lg U)$ bits that can return the number occ of points in any range $[i, j]$ in time $\mathcal{O}\left(\lg \lg \frac{j-i+1}{\text{occ}+1}\right)$.*

Proof. In Sections 2.4.1 and 2.4.2. \blacktriangleleft

2.4.1 Data structure

We use $\lceil \lg n \rceil - 1$ levels. At each level $\ell \geq 2$ we build a data structure that efficiently answers queries of length between $2^{\ell-2} + 1$ and $2^{\ell-1}$. Our structure defines specific *intervals* and *subintervals*. For clarity we will refer to *ranges* to denote any other range of the universe.

2.4.1.1 Data structure for level ℓ

Given a level ℓ , we divide the universe into $\lceil U/2^{\ell-1} \rceil$ overlapping *intervals* of size 2^ℓ , so that interval number k will be $[2^{\ell-1}k + 1, 2^{\ell-1}k + 2^\ell]$. It is clear that any range of size at most $2^{\ell-1}$ will be included in at least one interval.

We only consider non-empty intervals. We can have at most $2n$ non-empty intervals, as each point belongs to 2 intervals. We use a minimal perfect hash function f_ℓ that maps the $n' \leq 2n$ non-empty intervals into unique numbers in $[1, n']$. The minimal perfect hash function [17] uses $\mathcal{O}(n' + \lg \lg U) = \mathcal{O}(n + \lg \lg U)$ bits of space and works in constant time.

We consider how to solve queries on non-empty intervals. Suppose that an interval k contains n_k elements. We cut the interval into n_k equally-sized *subintervals*, of size $\lceil 2^\ell/n_k \rceil$ (the last subinterval can be shorter). We use a prefix sum data structure to store the number of elements in each subinterval of the interval k , by appending the cardinalities of the subintervals in a bitmap in unary and using select to get prefix sums. Such a prefix sum data structure uses $\mathcal{O}(n_k)$ bits. The space usage over all the prefix-sum data structures for all the intervals is $\mathcal{O}(n)$ bits. We concatenate the prefix-sum data structures of the intervals (in the order given by the minimal perfect hash function) and store another bitmap D_ℓ that marks the beginning of the prefix-sum data structure of each interval. This new bitmap also uses $\mathcal{O}(n)$ bits.

2.4.1.2 Global data structure

Our global data structure consists of the following pieces, adding up to $\mathcal{O}(n \lg U)$ bits.

1. The data structures of Section 2.4.1.1 built for levels $\ell = 2$ to $\ell = \lceil \lg n \rceil$. Each data structure uses $\mathcal{O}(n + \lg \lg U)$ bits of space, resulting in $\mathcal{O}(n \lg n + \lg n \lg \lg U)$ bits overall.
2. A *short-distance sensitive* predecessor data structure for the set of points that, given an element $x \in U$, returns the predecessor of x in $\mathcal{O}(\lg \lg d)$ time, where d is the minimum of the distances from x to its predecessor and to its successor in S . This data structure occupies $\mathcal{O}(n \lg U)$ bits of space. It was described by Bose et al. [8, 9], and the space was further improved recently by Belazzougui et al. [5].
3. A constant-time *range-emptiness* data structure for the set of points, by Alstrup et al. [1]. This data structure is given a range $[i, j]$ and returns whether the range $[i, j]$ contains at least one point or not. It also requires $\mathcal{O}(n \lg U)$ bits of space.

2.4.2 Query answering

We first do a range-emptiness query to determine whether the query range $[i, j]$ contains at least one element. If not, we immediately return 0. Otherwise we compute $\ell = \lceil \lg(j - i + 1) \rceil + 1$. Then, in constant time, we determine the interval of level ℓ that encloses $[i, j]$ and go to that level to answer the query. We note that interval by $[I, J]$.

We first use f_ℓ to find the index k of the interval $[I, J]$. Because we know that the interval is non-empty, we are sure that the minimal perfect hash function gives a meaningful answer. Next we use D_ℓ to recover the prefix-sum data structure for the interval $[I, J]$. Then, we

find the subinterval $[I_0, J_0]$ of $[I, J]$ that contains i and the subinterval $[I_1, J_1]$ that contains j . Notice that the count of the number of elements in $[i, j]$ equals the sum of the count of the number of elements in the three ranges $[i, J_0]$, $[J_0 + 1, I_1 - 1]$ and $[I_1, j]$. The count of the ranges $[J_0 + 1, I_1 - 1]$ can be found in constant time using the prefix sum data structure associated to interval $[I, J]$, as the range $[J_0 + 1, I_1 - 1]$ is aligned to subinterval boundaries.

What remains is to determine the counts in the two tail ranges $[i, J_0]$ and $[I_1, j]$. We only show how to determine the count in range $[i, J_0]$; the other case is symmetric. First we query the prefix-sum data structure in order to determine whether the subinterval $[J_0 - \lceil 2^\ell/n_k \rceil + 1, J_0]$ is empty or not. If the subinterval is empty, then we conclude that the range $[i, J_0]$ is also empty. Otherwise the subinterval $[J_0 - \lceil 2^\ell/n_k \rceil + 1, J_0]$ contains at least one element, so we use the distance-sensitive predecessor data structure to count the number of elements in $[i, J_0]$. That is, we call it to compute the predecessors of i and J_0 respectively and compute count of elements by subtracting the rank of the predecessor of i from the rank of the predecessor of J_0 . The query time of the distance sensitive data structure will be $\mathcal{O}(\lg \lg \lceil 2^\ell/n_k \rceil)$, since the distance of any element in the subinterval to i and to J_0 is at most $\lceil 2^\ell/n_k \rceil$. Now note that n_k , the number of elements in $[I, J]$, is at least occ . On the other hand $J - I + 1 = 2^\ell$ is at most $4(j - i + 1)$. We thus conclude that the query time is $\mathcal{O}\left(\lg \lg \left(\frac{j-i+1}{occ+1}\right)\right)$, and Theorem 9 is proved.

Given that we are building on an unpublished result [5], we include in Appendix C a data structure that needs $o(n \lg U)$ bits on top of the original array and answers predecessor queries for an element $y \in [x, z]$, limited to the range $[x, z]$ and assuming such range is not empty, in time $\mathcal{O}(\lg \lg(z - x + 1))$. Our result in this section is also obtained with this simpler data structure, applied on the interval given by $x = J_0 - \lceil 2^\ell/n_k \rceil + 1$ and $z = J_0$.

2.4.3 Back to the original problem

The next result, also of independent interest, is the key to improve our time complexity.

► **Theorem 10.** *We can store A in $\mathcal{O}(n)$ words so that later, given i, j and a , we can return the number occ of occurrences of element a in $A[i, j]$ in time $\mathcal{O}\left(\lg \lg \frac{j-i+1}{occ+1}\right)$.*

Proof. For each distinct element $a \in [1, \sigma]$ of A we store the positions where a occurs in $A[1, n]$ using the data structure of Theorem 9. If a occurs n_a times, we use $\mathcal{O}(n_a \lg n)$ bits, for a total of $\mathcal{O}(n \lg n)$ bits, or $\mathcal{O}(n)$ words. We also need $\mathcal{O}(\sigma) = \mathcal{O}(n)$ words for the pointers to each data structure. To answer a query that asks for the number of occurrences of a in $[i, j]$, we use the data structure for a and answer the corresponding range counting query. ◀

In our specific PRMaj problem we have an interval $[i, j]$ and have a specific element a that occurs at least once in $[i, j]$. Note that the element we want to count is marked only if it appears at least $2^b/2^t$ times in $[i - 2^{b+1}, j + 2^{b+1}]$. This interval is of size at most 2^{b+3} and contains at least $2^b/2^t$ elements. Thus to count the number of elements in $[i, j]$, we can use level $\ell = b + 4$ so that we can be sure to find an interval that fully covers $[i - 2^{b+1}, j + 2^{b+1}]$. The counting time will be $\mathcal{O}\left(\lg \lg \left\lceil \frac{2^{b+4}}{2^b/2^t} \right\rceil\right) = \lg \lg(2^4 2^t) = \mathcal{O}(\lg t) = \mathcal{O}(\lg \lg(1/\tau))$. Thus we have obtained our result.

► **Theorem 11.** *We can store A in $\mathcal{O}(n)$ words such that later, given i, j and τ , in time $\mathcal{O}((1/\tau) \lg \lg(1/\tau))$ we can list all the distinct elements that occur at least $\tau(j - i + 1)$ times in $A[i..j]$.*

3 Parameterized Range Minority

The main idea in this section, following Chan et al. [10], is to find sufficiently many different elements in the interval so that one of them must be a minority, if there is one. The problem reduces to using appropriate data structures to compute the frequency of the candidates in the query range using rank queries.

3.1 Nearly compressed space and no slowdown

Recall from Section 2.2 that C is the array in which $C[k] = \text{select}_{A[k]}(\text{rank}_{A[k]}(k) - 1)$ if $A[k]$ is not the first occurrence of that distinct element in A , or 0 if it is. Therefore, if we have a data structure that supports access, partial rank and select on A in $\mathcal{O}(f(n))$ time, then we can support access to C in $\mathcal{O}(f(n))$ time. Note that Muthukrishnan's [21] technique to find the distinct elements in $A[i..j]$ is real-time, that is, it gives the first κ answers in time $\mathcal{O}(\kappa)$, for any κ . Combining these observations yields the following lemma.

► **Lemma 12.** *Suppose we have data structures that support access, partial rank and select on A and RMQs on C , all in $\mathcal{O}(f(n))$ time. Then, given i and j , we can list the positions of the first occurrences in $A[i..j]$ of κ distinct elements in $A[i..j]$ using $\mathcal{O}(f(n))$ worst-case time per position listed, for any κ .*

Belazzougui and Navarro [6] showed how we can store a set of monotone minimal perfect hash functions in $\mathcal{O}(n) + o(nH)$ bits and support partial rank on A in $\mathcal{O}(1)$ time. Barbay et al. [2] showed how, given any constant $\epsilon > 0$, we can store A in $(1 + \epsilon)nH + o(n)$ bits such that we can support access and select on A in $\mathcal{O}(1)$ time. This yields the next result.

► **Theorem 13.** *Given any constant $\epsilon > 0$, we can store A in $(1 + \epsilon)nH + \mathcal{O}(n)$ bits such that later, given i, j and τ , in $\mathcal{O}(1/\tau)$ time we can find an element that occurs in $A[i..j]$ but at most $\tau(j - i + 1)$ times, or determine that no such element exists.*

Proof. We use Belazzougui and Navarro's and Barbay et al.'s data structures to store A in $(1 + \epsilon)nH + \mathcal{O}(n)$ bits such that we can support access, partial rank and select in $\mathcal{O}(1)$ time. We store an instance of Fischer's RMQ data structure [14], which takes $\mathcal{O}(n)$ bits, so that we can support RMQs on C in $\mathcal{O}(1)$ time. Given i, j and τ , we use Lemma 12 to list the positions of the first occurrences in $A[i..j]$ of $\lceil 1/\tau \rceil$ distinct elements (or all of them, if there are fewer). Notice there cannot be $\lceil 1/\tau \rceil$ distinct elements that all occur more than $\tau(j - i + 1)$ times in $A[i..j]$. For each position k in this list, we check whether $A.\text{select}_{A[k]}(A.\text{rank}_{A[k]}(k) + \lceil \tau(j - i + 1) \rceil - 1) \geq j$ and, if so, return $A[k]$ and stop. This takes a total of $\mathcal{O}(1/\tau)$ time. ◀

3.2 Compressed space and nearly no slowdown

In another paper, Belazzougui and Navarro [7] showed how we can store A in $nH + o(n(H+1))$ bits and support access and select in $\mathcal{O}(f(n))$ time for any function $f(n) = \omega(1)$. If we use this result in the proof of Theorem 13 instead of Barbay et al.'s, with $f(n) = \alpha(n)$, then we obtain the new version below.

► **Theorem 14.** *We can store A in $nH + \mathcal{O}(n) + o(nH)$ bits such that later, given i, j and τ , in $\mathcal{O}(\alpha(n)/\tau)$ time we can find an element that occurs in $A[i..j]$ but at most $\tau(j - i + 1)$ times, or determine that no such element exists.*

In the full version of this paper we will reduce the space bound in Theorem 14 to $nH + o(n(H + 1))$ bits. For those reviewers who are curious, we have included a sketch of the ideas in Appendix B.

4 Conclusions

We have given the first linear-space data structure with $\mathcal{O}(1/\tau)$ query time, which is worst-case optimal. Moreover, we have improved the space bounds for parameterized range majority and minority in the important case of variable τ . For PRMaj, we can achieve nearly linear space with optimal time, or up to compressed space with a slight slowdown. For PRMin, we can achieve nearly compressed space with time $\mathcal{O}(1/\tau)$, or compressed space with a slight slowdown. In the full version of this paper we will show how to use fully compressed space with the same slowdowns. The slowdowns are quite small — by factors of $\mathcal{O}(\lg \lg \sigma)$ to $\mathcal{O}(\alpha(n))$ — but we would like to eliminate them entirely, or show that this is impossible. We leave that as the main open problem from this paper.

Acknowledgments

Many thanks to Patrick Nicholson for his valuable comments.

References

- 1 S. Alstrup, G. Brodal, and T. Rauhe. Optimal static range reporting in one dimension. In *Proceedings of the 33rd ACM Symposium on Theory of Computing (STOC)*, pages 476–482, 2001.
- 2 J. Barbay, F. Claude, T. Gagie, G. Navarro, and Y. Nekrich. Efficient fully-compressed sequence representations. Technical Report 0911.4981v4, arxiv.org, 2012.
- 3 J. Barbay, T. Gagie, G. Navarro, and Y. Nekrich. Alphabet partitioning for compressed rank/select and applications. In *Proceedings of the 21st International Symposium on Algorithms and Computation (ISAAC)*, pages 315–326, 2010.
- 4 D. Belazzougui, P. Boldi, R. Pagh, and S. Vigna. Theory and practise of monotone minimal perfect hashing. In *Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 132–144, 2009.
- 5 D. Belazzougui, P. Boldi, and S. Vigna. Succinct indexes for predecessor search with distance-sensitive query times. Unpublished.
- 6 D. Belazzougui and G. Navarro. Alphabet-independent compressed text indexing. In *Proceedings of the 19th European Symposium on Algorithms (ESA)*, pages 748–759, 2011.
- 7 D. Belazzougui and G. Navarro. New lower and upper bounds for representing sequences. In *Proceedings of the 20th European Symposium on Algorithms (ESA)*, volume LNCS 7501, pages 181–192, 2012.
- 8 P. Bose, K. Douïeb, V. Dujmovic, J. Howat, and P. Morin. Fast local searches and updates in bounded universes. In *Proceedings of the 22nd Canadian Conference on Computational Geometry (CCCG)*, pages 261–264, 2010.
- 9 P. Bose, K. Douïeb, V. Dujmovic, J. Howat, and P. Morin. Fast local searches and updates in bounded universes. *Computational Geometry*, 2012. Advance access, DOI 10.1016/j.comgeo.2012.01.002.
- 10 T. M. Chan, S. Durocher, M. Skala, and B. T. Wilkinson. Linear-space data structures for range minority query in arrays. In *Proceedings of the 13th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT)*, pages 295–306, 2012.

- 11 S. Durocher, M. He, J. I. Munro, P. K. Nicholson, and Matthew Skala. Range majority in constant time and linear space. In *Proceedings of the 38th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 244–255, 2011.
- 12 A. Elmasry, J. I. Munro, and P. K. Nicholson. Dynamic range majority data structures. In *Proceedings of the 22nd International Symposium on Algorithms and Computation (ISAAC)*, pages 150–159, 2011.
- 13 P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms*, 3(3), 2007.
- 14 J. Fischer. Optimal succinctness for range minimum queries. In *Proceedings of the 9th Latin American Symposium on Theoretical Informatics (LATIN)*, pages 158–169, 2010.
- 15 T. Gaggie, M. He, J. I. Munro, and P. K. Nicholson. Finding frequent elements in compressed 2D arrays and strings. In *Proceedings of the 18th Symposium on String Processing and Information Retrieval (SPIRE)*, pages 295–300, 2011.
- 16 A. Golynski, J. I. Munro, and S. S. Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proceedings of the 17th Symposium on Discrete Algorithms (SODA)*, pages 368–373, 2006.
- 17 T. Hagerup and T. Tholey. Efficient minimal perfect hashing in nearly minimal space. In *Proceedings of the 18th International Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 317–326, 2001.
- 18 M. Karpinski and Y. Nekrich. Searching for frequent colors in rectangles. In *Proceedings of the 20th Canadian Conference on Computational Geometry (CCCG)*, pages 11–14, 2008.
- 19 Y. K. Lai, C. K. Poon, and B. Shi. Approximate colored range and point enclosure queries. *Journal of Discrete Algorithms*, 6(3):420–432, 2008.
- 20 J. Misra and D. Gries. Finding repeated elements. *Science of Computer Programming*, 2(2):143–152, 1982.
- 21 S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proceedings of the 13th Symposium on Discrete Algorithms (SODA)*, pages 657–666, 2002.
- 22 M. Pătraşcu. Succincter. In *Proceedings of the 49th Symposium on Foundations of Computer Science (FOCS)*, pages 305–313, 2008.
- 23 K. Sadakane. Succinct data structures for flexible text retrieval systems. *Journal of Discrete Algorithms*, 5(1):12–22, 2007.
- 24 Z. Wei and K. Yi. Beyond simple aggregates: indexing for summary queries. In *Proceedings of the 30th Symposium on Principles of Database Systems (PODS)*, pages 117–128, 2011.

■■■■■ Appendices included for review purposes only. ■■■■■

A Sketch of Fully-Compressed Parameterized Range Majority

As in the proof of Theorem 8, we start by storing A in a data structure by Barbay et al. [3, 2] that takes $nH + o(n(H + 1))$ bits and supports access in constant time, and rank and select in $\mathcal{O}(\lg \lg \sigma)$ time. We can assume $\sigma = \omega(1)$ since, otherwise, we can compute the frequency of each distinct symbol in a given range, in a total of $\mathcal{O}(\sigma \lg \lg \sigma) = \mathcal{O}(1)$ time. Barbay et al.’s data structure is based on partitioning the set of distinct elements in A according to their frequencies and storing data structures supporting access, rank and select on, first, an array $T[1..n]$ with $T[k] = \ell$ indicating that we place $A[k]$ in the ℓ th subset; second, for $1 \leq \ell \leq \lceil \lg^2 n \rceil$, an array S_ℓ with $S_\ell[k]$ indicating the rank of $A[T.\text{select}_\ell(k)]$ in the ℓ th subset.

We add data structures such that we can support PRMaj queries on T and each S_ℓ . Let A' be either T or an array S_ℓ , let n' be the length of A' and let $\sigma' \leq \sigma$ be the number of distinct elements in A' . In the full version of this paper we will prove, by generalizing

Corollary 7, that we can store $\mathcal{O}\left(n' \lg \sigma' \cdot \frac{\lg \lg \lg \sigma}{\lg \lg \sigma} + \frac{n'}{\lg n'}\right)$ more bits such that we can answer PRMaj queries on A' in $\mathcal{O}((1/\tau) \lg \lg \sigma)$ time. For now, however, we use a clumsier but easier approach.

If A' is T or $\sigma' \leq \lg n$ then, as part of Barbay et al.'s data structure, we have A' stored as a multiary wavelet tree that supports access, rank and select in $\mathcal{O}(1)$ time; otherwise, we have it stored as an instance of a data structure by Golynski, Munro and Rao [16] that supports access in $\mathcal{O}(1)$ time and rank and select in $\mathcal{O}(\lg \lg \sigma)$ time. Therefore, if $\sigma' \leq \lg \lg \sigma$, we can compute the frequency of each distinct symbol in any given range, in a total of $\mathcal{O}(\sigma') = \mathcal{O}(\lg \lg \sigma)$ time. If $\sigma' > \lg \lg \sigma$, then we apply Corollary 7 to A' for values of t between 0 and $\lceil \lg \sigma' \rceil$. This way, we store

$$\mathcal{O}\left(\frac{n' \lg \sigma' \lg \lg \lg \sigma'}{\lg \lg \sigma'} + \frac{n'}{\lg n'}\right) = \mathcal{O}\left(n' \lg \sigma' \cdot \frac{\lg \lg \lg \lg \lg \sigma}{\lg \lg \lg \lg \sigma} + \frac{n'}{\lg n'}\right)$$

bits more bits for A' and can answer PRMaj queries on it in $\mathcal{O}((1/\tau) \lg \lg \sigma')$ time.

Given i, j and τ , we use a PRMaj query on T to find the $\mathcal{O}(1/\tau)$ distinct elements that each occur at least $\tau(j-i+1)$ times in $T[i..j]$. For each such element ℓ , we compute $i_\ell = T.\text{rank}_\ell(i-1)+1$, $j_\ell = T.\text{rank}_\ell(j)$ and $\tau_\ell = \frac{\tau(j-i+1)}{j_\ell - i_\ell + 1}$. We then use a PRMaj query on S_ℓ to find the $\mathcal{O}(1/\tau_\ell)$ distinct elements that each occur at least $\tau_\ell(j_\ell - i_\ell + 1)$ times in $S_\ell[i_\ell..j_\ell]$. We then map each of these distinct elements to the corresponding distinct element in A using another multiary wavelet tree that is part of Barbay et al.'s data structure. All these operations together take a total of $\mathcal{O}((1/\tau + \sum_\ell 1/\tau_\ell) \lg \lg \sigma)$ time. Since $\sum_\ell (j_\ell - i_\ell + 1) \leq j - i + 1$, we have $\sum_\ell 1/\tau_\ell \leq 1/\tau$. Therefore, we use a total of $\mathcal{O}((1/\tau) \lg \lg \sigma)$ time.

► **Theorem 15.** *We can store A in $nH + o(n(H+1))$ bits such that later, given i, j and τ , in $\mathcal{O}((1/\tau) \lg \lg \sigma)$ time we can list all the distinct elements that occur at least $\tau(j-i+1)$ times in $A[i..j]$. The structure also offers constant-time access to A and $\mathcal{O}(\lg \lg \sigma)$ -time rank and select.*

B Sketch of Fully-Compressed Parameterized Range Minority

The $\mathcal{O}(n)$ term in the space bound of Theorem 14 comes from two places: first, Belazzougui and Navarro's [6] set of monotone minimal perfect hash functions, which take $\mathcal{O}(n) + o(nH)$ bits; second, Fischer's [14] RMQ data structure, which takes $2n + o(n)$ bits. To eliminate that term, we first show that we need only $o(n(H+1))$ bits for the monotone minimal perfect hash functions, and then show how to build Fischer's data structure over a sampling of C .

Belazzougui and Navarro's [7] data structure supporting access and rank on A is also based on alphabet partitioning (see Appendix A). This means their data structure supports access, rank and select queries in $\mathcal{O}(1)$ time whenever the element involved occurs at least $n/\lg n$ times in A . It follows that we need store monotone minimal perfect hash functions only for elements occurring fewer than $n/\lg n$ times in A , and calculation shows this takes a total of $o(n(H+1))$ bits. We will provide more details in the full version of this paper.

► **Theorem 16.** *We can store A in $nH + o(n(H+1))$ bits and support access and select in $\mathcal{O}(f(n))$ time for any function $f(n) = \omega(1)$, and partial rank in $\mathcal{O}(1)$ time.*

Applying Theorem 16 to A lets us support access to C in $\mathcal{O}(\sqrt{\alpha(n)})$ time. Let C' be the array defined by $C'[k] = \min\left(C\left[\left\lceil (k-1)\sqrt{\alpha(n)} \right\rceil + 1\right], \dots, C\left[\left\lceil k\sqrt{\alpha(n)} \right\rceil\right]\right)$, which has length $\mathcal{O}\left(n/\sqrt{\alpha(n)}\right) = o(n)$. We store an instance of Fischer's RMQ data structure for

C' in $o(n)$ bits. Given an interval in C , we use this data structure to find a subinterval of length $\sqrt{\alpha(n)}$ that contains that interval's leftmost minimum. We then compute the entries of C in the that subinterval and find the exact position of the leftmost minimum, in a total of $\mathcal{O}(\alpha(n))$ time.

► **Theorem 17.** *We can store A in $nH + o(n(H + 1))$ bits such that later, given i, j and τ , in $\mathcal{O}(\alpha(n)/\tau)$ time we can find an element that occurs in $A[i..j]$ but at most $\tau(j - i + 1)$ times, or determine that no such element exists.*

C A Simple Distance-Sensitive Data Structure

Assume we have a table X that contains n elements from universe U in sorted order.

We use $\lg \lg U$ levels. At a level ℓ , we divide the universe into $\lceil U/2^{2^\ell} \rceil$ overlapping intervals, so that interval k will be $[k \cdot 2^{2^\ell} + 1, (k+2)2^{2^\ell}]$. We consider separately the intervals with even and odd k (we call them odd or even intervals). For each of the two categories, the set of intervals will be disjoint. For each category, we use a monotone minimum perfect hash function (mmpfh) F_ℓ that stores the k values corresponding to nonempty intervals, and a prefix sum data structure to store the number of elements in each nonempty interval k , by appending the cardinalities of the intervals in a bitmap B_ℓ in unary (i.e., cardinality c is stored as $1^{c-1}0$) and using select to get prefix sums. With F_ℓ and B_ℓ we map in constant time from a nonempty interval k to its corresponding area in X (say, $p = F_\ell(k)$, then the area is $X[\text{select}_0(B_\ell, k - 1) + 1 .. \text{select}_0(B_\ell, k)]$). Since there are at most n nonempty intervals of each category, F_ℓ uses $\mathcal{O}(n \lg \lg U)$ bits. Bitmap B_ℓ uses $\mathcal{O}(n)$ bits.

In addition, for each nonempty interval with more than 2^ℓ elements, we store a local predecessor search data structure (lpsds). The lpsds of an interval samples one every 2^ℓ elements in the interval and stores them in a local y-fast trie. The y-fast trie of elements $x \in [k \cdot 2^{2^\ell} + 1, (k+2)2^{2^\ell}]$ will store $x - k \cdot 2^{2^\ell} - 1$, and thus will range over a universe of size $\mathcal{O}(2^{2^\ell})$. Since they store, in total, $\mathcal{O}(n/2^\ell)$ elements over a universe of size $\mathcal{O}(2^{2^\ell})$, the space of all the lpsds adds up to $\mathcal{O}(n)$ bits.

Since the lpsds storing m_r elements uses at most cm_r bits, for some constant c , we store them one after the other, reserving cm_r bits for each lpsds storing m_r elements. We store a bitmap P_ℓ as a partial sum data structure on the m_r values, concatenating $1^{m_r}0$ (assume $m_r = 0$ if there are $\leq 2^\ell$ elements and thus no lpsds is stored). We can find in constant time, using select on P_ℓ , the starting point of each lpsds. P_ℓ uses at most $\mathcal{O}(n)$ bits.

Then to do predecessor search on interval $[i, j]$ we proceed as follows:

1. We compute $\ell = \lceil \lg \lg(j - i + 1) \rceil$, so that the query is for sure contained in an (even or odd) interval k of level ℓ . Number k is found algebraically in constant time.
2. We use F_ℓ to map k to its position p in the nonempty intervals, and then B_ℓ to find the corresponding range $X[i_k..j_k]$. This takes constant time.
3. If $j_k - i_k + 1 \leq 2^\ell$, we complete the query with a binary search on $X[i_k..j_k]$, in time $\mathcal{O}(\ell)$, and finish.
4. We use the local predecessor search data structure of interval p , found using P_ℓ , to determine the subinterval $[i'_k..j'_k] \subseteq [i_k..j_k]$ of size 2^ℓ where the answer lies. This takes time $\mathcal{O}(\lg \lg 2^{2^\ell}) = \mathcal{O}(\ell)$.
5. We complete the query using binary search on $X[i'_k..j'_k]$, in time $\mathcal{O}(\ell)$.

Our data structures use in total $\mathcal{O}(n \lg \lg U)$ bits for a given level ℓ , which adds up to $\mathcal{O}(n(\lg \lg U)^2)$ bits in total. They answer in time $\mathcal{O}(\ell) = \mathcal{O}(\lg \lg(j - i + 1))$ on nonempty intervals. Note that on empty intervals our mmpfh F_ℓ could return an arbitrary value.